<https://leetcode.com/problems/powx-n/description/>

Implement pow(x, n), which calculates x raised to the power n (i.e., x^n).

**Example 1:**

Input: x = 2.00000, n = 10

Output: 1024.00000

**Example 2:**

Input: x = 2.10000, n = 3

Output: 9.26100

**Example 3:**

Input: x = 2.00000, n = -2

Output: 0.25000

Explanation: 2^-2 = 1/2^2 = 1/4 = 0.25

**Constraints:**

-100.0 < x < 100.0

-231 <= n <= 231-1

n is an integer.

Either x is not zero or n > 0.

-104 <= xn <= 104

**Attempt 1: 2024-11-29**

**Solution 1: Math (60 min)**

class Solution {

    public double myPow(double x, int n) {

        // If power n is negative, calculate the inverse of the power

        if(n < 0) {

            x = 1 / x;

            n = -n;

        }

        double result = 1;

        // Loop through all bits of the exponent 'n'

        while(n != 0) {

            // Must add () for n & 1 since the Operator Precedence

            // has '==' higher than '&'

            // If the current bit is set, multiply the result by the base

            if((n & 1) == 1) {

                result \*= x;

            }

            // Square the base for the next bit in the exponent

            x \*= x;

            n >>>= 1;

        }

        return result;

    }

}

Time Complexity: O(logn)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/powx-n/solutions/1337794/java-c-simple-o-log-n-easy-faster-than-100-explained/>

**FIRST APPROACH**

We can solve this problem by multiplying x by n times

eg:

x = 7 and n = 11

7 \* 7 \* 7 \* 7 \* 7 \* 7 \* 7 \* 7 \* 7 \* 7 \* 7 = 1977326743

Here we have multiplied 7 for 11 times, which will result in O(n)

But, Suppose x = 1 and n = 2147483647

If we follow this approach then, 1 will be multiplied 2147483647 times which is not efficient at all.

COMPLEXITY

Time: O(n), where n is the given power

Space: O(1), in-place

**SECOND APPROACH**

In order to improve efficiency we will opt for Binary Exponentiation using which we can calculate x^n using O log2(N) multiplications.

Basic Idea is to divide the work using binary representation of exponents

i.e. is to keep multiplying pow with x, if the bit is odd, and multiplying x with itself until we get 0

We will use very 1st example of 1st Approach i.e.

x = 7, n = 11 and pow = 1

Here, we have to calculate 7^11

Binary of n i.e. (11)10 is (1011)2

1 0 1 1

2^3 2^2 2^1 2^0 <-- Corresponding place values of each bit

OR we can also write this as

1 0 1 1

8 4 2 1 <-- Corresponding place values of each bit

Now, 7^8 × 7^2 × 7^1 == 7^11 as 7^(8 + 2 + 1) == 7^11

NOTE: We have not considered 7^4 in this case as the 4th place bit is OFF

So, 7^8 × 7^2 × 7^1 == 5764801 × 49 × 7 == 1977326743 <-- Desired Output

Now, applying logic keeping this concept in mind

double pow = 1;

while(n != 0){

if((n & 1) != 0) // equivalent to if((n % 2) != 0) i.e. multiply only when the number is odd

pow \*= x;

x \*= x;

n >>>= 1; // equivalent to n = n / 2; i.e. keep dividing the number by 2

}

**PROCESS**

**Iteration 1**

pow = 1 × 7 = 7

x = 7 × 7 = 49

n = 11 >>> 1 = 5

**Iteration 2**

pow = 7 × 49 = 343

x = 49 × 49 = 2401

n = 5 >>> 1 = 2

**Iteration 3**

x = 2401 × 2401 = 5764801

n = 2 >>> 1 = 1

**Iteration 4**

pow = 343 × 5764801 = 1977326743

x = 5764801 × 5764801 = 3.323293057 × 10^13

n = 1 >>> 1 = 0

We exit the loop as the number has become 0 and we got pow as 1977326743 which is the desired output

In this binary exponentiation approach, the loop iterated for only 4 times which is nothing but (O log2(N) + 1) ~ (O log2(N))

And for 2nd example of 1st Approach where

x = 1 and n = 2147483647

This loop executed for only 31 times (O log2(N)) which is far far less than 2147483647 times(in case of O(N) approach)

class Solution {

public double myPow(double x, int n) {

if(n < 0){

n = -n;

x = 1 / x;

}

double pow = 1;

while(n != 0){

if((n & 1) != 0){

pow \*= x;

}

x \*= x;

n >>>= 1;

}

return pow;

}

}

**COMPLEXITY**

Time: O(log2(n)), where n is the given power

Space: O(1), in-place

**Refer to**

<https://algo.monster/liteproblems/50>

**Problem Description**

The task here is to implement a function that calculates the result of raising a number x to the power of n. In mathematical terms, you're asked to compute x^n. This function should work with x as a float (meaning it can be a decimal number) and n as an integer (which can be positive, negative, or zero).

**Intuition**

The intuitive approach to solve exponentiation could be to multiply x by itself n times. However, this method is not efficient for large values of n as it would have linear complexity O(n). Instead, we can use the "fast power algorithm," or "binary exponentiation," which reduces the time complexity substantially to O(log n).

This optimized solution is based on the principle that for any number a and integer n, the power a^n can be broken down into smaller powers using the properties of exponents. Specifically, if n is even, a^n can be expressed as (a^(n/2))^2, and if n is odd, it can be written as a\*(a^((n-1)/2))^2. Applying these properties repeatedly allows us to reduce the problem into smaller subproblems, efficiently using divide-and-conquer strategy.

In this code, the function qpow implements this efficient algorithm using a loop, squaring a and halving n at each iteration, and multiplying to the answer when n is odd (detected by n & 1 which checks the least significant bit). The loop continues until n becomes 0, which is when we have multiplied enough factors into ans to give us our final result.

If the exponent n is negative, we calculate x^(-n) and then take its reciprocal, as by mathematical rule, x^(-n) equals 1/(x^n). This is why, in the final return statement, the solution checks if n is non-negative and either returns qpow(x, n) or 1 / qpow(x, -n) accordingly.

**Solution Approach**

The solution uses a helper function called qpow to implement a fast exponentiation algorithm iteratively. The function takes two parameters, a which represents the base number x, and n which represents the exponent. Here's a walkthrough of how the algorithm works:

Initialize an accumulator (ans): A variable ans is initialized to 1. This will eventually hold the final result after accumulating the necessary multiplications.

Iterative process: A while loop is set up to run as long as n is not zero. Each iteration of this loop represents a step in the exponentiation process.

Checking if n is odd (using bitwise AND): Within the loop, we check if the least significant bit of n is 1 by using n & 1. This is equivalent to checking if n is odd. If it is, we multiply the current ans by a because we know we need to include this factor in our product.

Doubling the base (a \*= a): After dealing with the possibility of n being odd, we square the base a by multiplying it by itself. This corresponds to the exponential property that a^n = (a^(n/2))^2 for even n.

Halving the exponent (using bitwise shift): We then halve the exponent n by right-shifting it one bit using n >>= 1. This is equivalent to integer division by 2. Squaring a and halving n is repeated until n becomes zero.

Handling negative exponents: After calculating the positive power using qpow, if the original exponent n was negative, we take the reciprocal of the result by returning 1 / qpow(x, -n). This handles cases where x should be raised to a negative power.

Data Structures: The solution does not use any complicated data structures; it relies on simple variables to hold intermediate results (ans, a).

Patterns: The iterative process exploits the divide-and-conquer paradigm, breaking the problem down into smaller subproblems of squaring the base and halving the exponent.

Algorithm Efficiency: Because of the binary exponentiation technique, the time complexity of the algorithm is O(log n), which is very efficient even for very large values of n.

**Example Walkthrough**

Let's walk through the iterative fast exponentiation technique using an example where we want to calculate 3^4, which is 3 raised to the power of 4.

Initialize accumulator (ans): Set ans = 1. This will hold our final result.

Assign base to a and exponent to n: We start with a = 3 and n = 4.

Begin iterative process: Since n is not zero, enter the while loop.

First iteration (n = 4):

Check if n is odd: n & 1 is 0, meaning n is even. Skip multiplying ans by a.

Square the base (a \*= a): a becomes 9.

Halve the exponent (n >>= 1): n becomes 2.

Second iteration (n = 2):

Check if n is odd: n & 1 is 0, still even. Skip multiplying ans by a.

Square the base (a \*= a): a becomes 81.

Halve the exponent (n >>= 1): n becomes 1.

Third iteration (n = 1):

Check if n is odd: n & 1 is 1, meaning n is odd. Multiply ans by a: ans becomes 81.

Square the base (a \*= a): Although a becomes 6561 we stop because the next step would make n zero.

Exponent 0 check: The while loop exits because n is now zero.

Since we never had a negative power, we don't need to consider taking the reciprocal of ans, and the final answer for 3^4 is in ans, which is 81.

In this example, you can see we only had to iterate 3 times to compute 3^4, which is more efficient than multiplying 3 by itself 4 times. The optimized algorithm has a significant advantage as the values of n increase in magnitude.

**Solution Implementation**

class Solution {

public double myPow(double x, int n) {

// If power n is non-negative, calculate power using helper method

if (n >= 0) {

return quickPow(x, n);

} else {

// If power n is negative, calculate the inverse of the power

return 1 / quickPow(x, -(long) n);

}

}

private double quickPow(double base, long exponent) {

double result = 1; // Initialize result to neutral element for multiplication

// Loop through all bits of the exponent

while (exponent > 0) {

// If the current bit is set, multiply the result by the base

if ((exponent & 1) == 1) {

result \*= base;

}

// Square the base for the next bit in the exponent

base \*= base;

// Shift exponent to the right to process the next bit

exponent >>= 1;

}

// Return the final result of base raised to the exponent

return result;

}

}

**Time and Space Complexity**

The given code implements the binary exponentiation algorithm to calculate x raised to the power n.

**Time Complexity**

The time complexity of the algorithm is O(log n). This is because the while loop runs for the number of bits in the binary representation of n. In each iteration, n is halved by the right shift operation (n >>= 1), which reduces the number of iterations to the logarithm of n, hence O(log n).

**Space Complexity**

The space complexity of the algorithm is O(1). The function qpow only uses a constant amount of additional space for variables ans and a, which are used to keep track of the cumulative product and the base that's being squared in each iteration. There is no use of any data structure that grows with the input size, so the space requirement remains constant irrespective of n.